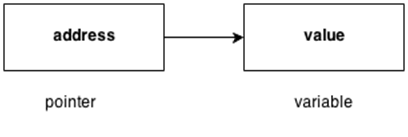
**C++ Pointers**

**The pointer in C++ language is a variable, it is also known as locator or indicator that points to an address of a value.**

****

**Advantage of pointer**

**1) Pointer reduces the code and improves the performance, it is used to retrieving strings, trees etc. and used with arrays, structures and functions.**

**2) We can return multiple values from function using pointer.**

**Java Try Catch**

**3) It makes you able to access any memory location in the computer's memory.**

**Usage of pointer**

**There are many usage of pointers in C++ language.**

**1) Dynamic memory allocation**

**In c language, we can dynamically allocate memory using malloc() and calloc() functions where pointer is used.**

**2) Arrays, Functions and Structures**

**Pointers in c language are widely used in arrays, functions and structures. It reduces the code and improves the performance.**

**Symbols used in pointer**

|  |  |  |
| --- | --- | --- |
| **Symbol** | **Name** | **Description** |
| **& (ampersand sign)** | **Address operator** | **Determine the address of a variable.** |
| **∗ (asterisk sign)** | **Indirection operator** | **Access the value of an address.** |

**Declaring a pointer**

**The pointer in C++ language can be declared using ∗ (asterisk symbol).**

1. **int ∗   a; //pointer to int**
2. **char ∗  c; //pointer to char**

**Pointer Example**

**Let's see the simple example of using pointers printing the address and value.**

1. **#include <iostream>**
2. **using namespace std;**
3. **int main()**
4. **{**
5. **int number=30;**
6. **int ∗   p;**
7. **p=&number;//stores the address of number variable**
8. **cout<<"Address of number variable is:"<<&number<<endl;**
9. **cout<<"Address of p variable is:"<<p<<endl;**
10. **cout<<"Value of p variable is:"<<\*p<<endl;**
11. **return 0;**
12. **}**

**Output:**

**Address of number variable is:0x7ffccc8724c4**

**Address of p variable is:0x7ffccc8724c4**

**Value of p variable is:30**

**Pointer Program to swap 2 numbers without using 3rd variable**

1. **#include <iostream>**
2. **using namespace std;**
3. **int main()**
4. **{**
5. **int a=20,b=10,∗p1=&a,∗p2=&b;**
6. **cout<<"Before swap: ∗p1="<<∗p1<<" ∗p2="<<∗p2<<endl;**
7. **∗p1=∗p1+∗p2;**
8. **∗p2=∗p1-∗p2;**
9. **∗p1=∗p1-∗p2;**
10. **cout<<"After swap: ∗p1="<<∗p1<<" ∗p2="<<∗p2<<endl;**
11. **return 0;**
12. **}**

**Output:**

**Before swap: ∗p1=20 ∗p2=10**

**After swap: ∗p1=10 ∗p2=20**

# C++ Array of Pointers

**Array and pointers are closely related to each other. In C++, the name of an array is considered às a pointer, i.e., the name of an array contains the address of an element. C++ considers the array name as the address of the first element. For example, if we create an array, i.e., marks which hold the 20 values of integer type, then marks will contain the address of first element, i.e., marks[0]. Therefore, we can say that array name (marks) is a pointer which is holding the address of the first element of an array.**

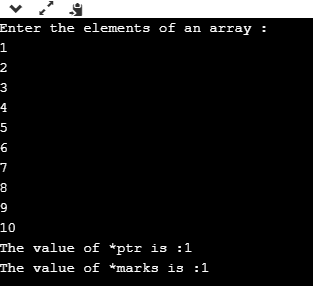
**Let's understand this scenario through an example.**

1. **#include <iostream>**
2. **using namespace std;**
3. **int main()**
4. **{**
5. **int \*ptr;  // integer pointer declaration**
6. **int marks[10]; // marks array declaration**
7. **std::cout << "Enter the elements of an array :" << std::endl;**
8. **for(int i=0;i<10;i++)**
9. **{**
10. **cin>>marks[i];**
11. **}**
12. **ptr=marks; // both marks and ptr pointing to the same element..**
13. **std::cout << "The value of \*ptr is :" <<\*ptr<< std::endl;**
14. **std::cout << "The value of \*marks is :" <<\*marks<<std::endl;**
15. **}**

**In the above code, we declare an integer pointer and an array of integer type. We assign the address of marks to the ptr by using the statement ptr=marks; it means that both the variables 'marks' and 'ptr' point to the same element, i.e., marks[0]. When we try to print the values of \*ptr and \*marks, then it comes out to be same. Hence, it is proved that the array name stores the address of the first element of an array.**

**Output**

**How to find Nth Highest Salary in SQL**

****

### Array of Pointers

**An array of pointers is an array that consists of variables of pointer type, which means that the variable is a pointer addressing to some other element. Suppose we create an array of pointer holding 5 integer pointers; then its declaration would look like:**

1. **int \*ptr[5];         // array of 5 integer pointer.**

**In the above declaration, we declare an array of pointer named as ptr, and it allocates 5 integer pointers in memory.**

**The element of an array of a pointer can also be initialized by assigning the address of some other element. Let's observe this case through an example.**

1. **int a; // variable declaration.**
2. **ptr[2] = &a;**

**In the above code, we are assigning the address of 'a' variable to the third element of an array 'ptr'.**

**We can also retrieve the value of 'a' be dereferencing the pointer.**

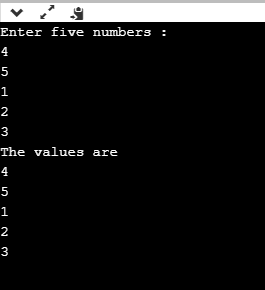
1. **\*ptr[2];**

**Let's understand through an example.**

1. **#include <iostream>**
2. **using namespace std;**
3. **int main()**
4. **{**
5. **int ptr1[5]; // integer array declaration**
6. **int \*ptr2[5]; // integer array of pointer declaration**
7. **std::cout << "Enter five numbers :" << std::endl;**
8. **for(int i=0;i<5;i++)**
9. **{**
10. **std::cin >> ptr1[i];**
11. **}**
12. **for(int i=0;i<5;i++)**
13. **{**
14. **ptr2[i]=&ptr1[i];**
15. **}**
16. **// printing the values of ptr1 array**
17. **std::cout << "The values are" << std::endl;**
18. **for(int i=0;i<5;i++)**
19. **{**
20. **std::cout << \*ptr2[i] << std::endl;**
21. **}**
22. **}**

**In the above code, we declare an array of integer type and an array of integer pointers. We have defined the 'for' loop, which iterates through the elements of an array 'ptr1', and on each iteration, the address of element of ptr1 at index 'i' gets stored in the ptr2 at index 'i'.**

**Output**

****

**Till now, we have learnt the array of pointers to an integer. Now, we will see how to create the array of pointers to strings.**

### Array of Pointer to Strings

**An array of pointer to strings is an array of character pointers that holds the address of the first character of a string or we can say the base address of a string.**

**The following are the differences between an array of pointers to string and two-dimensional array of characters:**

* **An array of pointers to string is more efficient than the two-dimensional array of characters in case of memory consumption because an array of pointer to strings consumes less memory than the two-dimensional array of characters to store the strings.**
* **In an array of pointers, the manipulation of strings is comparatively easier than in the case of 2d array. We can also easily change the position of the strings by using the pointers.**

**Let's see how to declare the array of pointers to string.**

**First, we declare the array of pointer to string:**

1. **char \*names[5] = {"john",**
2. **"Peter",**
3. **"Marco",**
4. **"Devin",**
5. **"Ronan"};**

**In the above code, we declared an array of pointer names as 'names' of size 5. In the above case, we have done the initialization at the time of declaration, so we do not need to mention the size of the array of a pointer. The above code can be re-written as:**

1. **char \*names[ ] = {"john",**
2. **"Peter",**
3. **"Marco",**
4. **"Devin",**
5. **"Ronan"};**

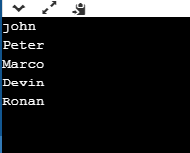
**In the above case, each element of the 'names' array is a string literal, and each string literal would hold the base address of the first character of a string. For example, names[0] contains the base address of "john", names[1] contains the base address of "Peter", and so on. It is not guaranteed that all the string literals will be stored in the contiguous memory location, but the characters of a string literal are stored in a contiguous memory location.**

**Let's create a simple example.**

1. **#include <iostream>**
2. **using namespace std;**
3. **int main()**
4. **{**
5. **char \*names[5] = {"john",**
6. **"Peter",**
7. **"Marco",**
8. **"Devin",**
9. **"Ronan"};**
10. **for(int i=0;i<5;i++)**
11. **{**
12. **std::cout << names[i] << std::endl;**
13. **}**
14. **return 0;**
15. **}**

**In the above code, we have declared an array of char pointer holding 5 string literals, and the first character of each string is holding the base address of the string.**

**Output**

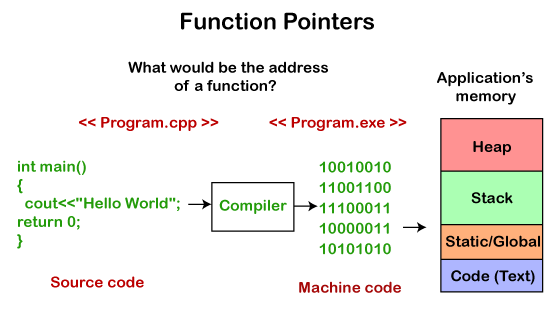
****

# Function Pointer in C++

**As we know that pointers are used to point some variables; similarly, the function pointer is a pointer used to point functions. It is basically used to store the address of a function. We can call the function by using the function pointer, or we can also pass the pointer to another function as a parameter.**

**They are mainly useful for event-driven applications, callbacks, and even for storing the functions in arrays.**

### What is the address of a function?

****

**Computer only understands the low-level language, i.e., binary form. The program we write in C++ is always in high-level language, so to convert the program into binary form, we use compiler. Compiler is a program that converts source code into an executable file. This executable file gets stored in RAM. The CPU starts the execution from the main() method, and it reads the copy in RAM but not the original file.**

**All the functions and machine code instructions are data. This data is a bunch of bytes, and all these bytes have some address in RAM. The function pointer contains RAM address of the first instruction of a function.**

**Difference between JDK, JRE, and JVM**

### Syntax for Declaration

**The following is the syntax for the declaration of a function pointer:**

1. **int (\*FuncPtr) (int,int);**

**The above syntax is the function declaration. As functions are not simple as variables, but C++ is a type safe, so function pointers have return type and parameter list. In the above syntax, we first supply the return type, and then the name of the pointer, i.e., FuncPtr which is surrounded by the brackets and preceded by the pointer symbol, i.e., (\*). After this, we have supplied the parameter list (int,int). The above function pointer can point to any function which takes two integer parameters and returns integer type value.**

### Address of a function

**We can get the address of a function very easily. We just need to mention the name of the function, we do not need to call the function.**

**Let's illustrate through an example.**

1. **#include <iostream>**
2. **using namespace std;**
3. **int main()**
4. **{**
5. **std::cout << "Address of a main() function is : " <<&main<< std::endl;**
6. **return 0;**
7. **}**

**In the above program, we are displaying the address of a main() function. To print the address of a main() function, we have just mentioned the name of the function, there is no bracket not parameters. Therefore, the name of the function by itself without any brackets or parameters means the address of a function.**

**We can use the alternate way to print the address of a function, i.e., &main.**

### Calling a function indirectly

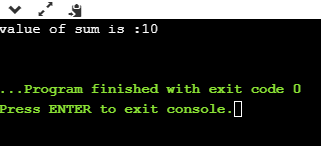
**We can call the function with the help of a function pointer by simply using the name of the function pointer. The syntax of calling the function through the function pointer would be similar as we do the calling of the function normally.**

**Let's understand this scenario through an example.**

1. **#include <iostream>**
2. **using namespace std;**
3. **int add(int a , int b)**
4. **{**
5. **return a+b;**
6. **}**
7. **int main()**
8. **{**
9. **int (\*funcptr)(int,int);  // function pointer declaration**
10. **funcptr=add; // funcptr is pointing to the add function**
11. **int sum=funcptr(5,5);**
12. **std::cout << "value of sum is :" <<sum<< std::endl;**
13. **return 0;**
14. **}**

**In the above program, we declare the function pointer, i.e., int (\*funcptr)(int,int) and then we store the address of add() function in funcptr. This implies that funcptr contains the address of add() function. Now, we can call the add() function by using funcptr. The statement funcptr(5,5) calls the add() function, and the result of add() function gets stored in sum variable.**

**Output:**

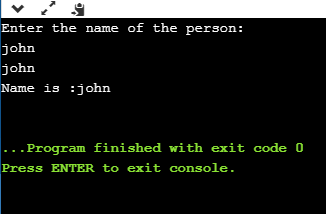
****

**Let's look at another example of function pointer.**

1. **#include <iostream>**
2. **using namespace std;**
3. **void printname(char \*name)**
4. **{**
5. **std::cout << "Name is :" <<name<< std::endl;**
6. **}**
8. **int main()**
9. **{**
10. **char s[20];  // array declaration**
11. **void (\*ptr)(char\*);  // function pointer declaration**
12. **ptr=printname;  // storing the address of printname in ptr.**
13. **std::cout << "Enter the name of the person: " << std::endl;**
14. **cin>>s;**
15. **cout<<s;**
16. **ptr(s);  // calling printname() function**
17. **return 0;**
18. **}**

**In the above program, we define the function printname() which contains the char pointer as a parameter. We declare the function pointer, i.e., void (\*ptr)(char\*). The statement ptr=printname means that we are assigning the address of printname() function to ptr. Now, we can call the printname() function by using the statement ptr(s).**

**Output:**

****

### Passing a function pointer as a parameter

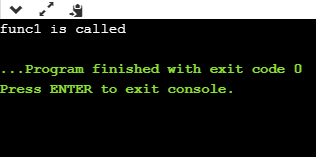
**The function pointer can be passed as a parameter to another function.**

**Let's understand through an example.**

1. **#include <iostream>**
2. **using namespace std;**
3. **void func1()**
4. **{**
5. **cout<<"func1 is called";**
6. **}**
7. **void func2(void (\*funcptr)())**
8. **{**
9. **funcptr();**
10. **}**
11. **int main()**
12. **{**
13. **func2(func1);**
14. **return 0;**
15. **}**

**In the above code, the func2() function takes the function pointer as a parameter. The main() method calls the func2() function in which the address of func1() is passed. In this way, the func2() function is calling the func1() indirectly.**

**Output:**

****

# What is Memory Management?

**Memory management is a process of managing computer memory, assigning the memory space to the programs to improve the overall system performance.**

## Why is memory management required?

**As we know that arrays store the homogeneous data, so most of the time, memory is allocated to the array at the declaration time. Sometimes the situation arises when the exact memory is not determined until runtime. To avoid such a situation, we declare an array with a maximum size, but some memory will be unused. To avoid the wastage of memory, we use the new operator to allocate the memory dynamically at the run time.**

## Memory Management Operators

**In C language, we use the malloc() or calloc() functions to allocate the memory dynamically at run time, and free() function is used to deallocate the dynamically allocated memory. C++ also supports these functions, but C++ also defines unary operators such as new and delete to perform the same tasks, i.e., allocating and freeing the memory.**

### New operator

**A new operator is used to create the object while a delete operator is used to delete the object. When the object is created by using the new operator, then the object will exist until we explicitly use the delete operator to delete the object. Therefore, we can say that the lifetime of the object is not related to the block structure of the program.**

**C++ vs Java**

**Syntax**

1. **pointer\_variable = new data-type**

**The above syntax is used to create the object using the new operator. In the above syntax, 'pointer\_variable' is the name of the pointer variable, 'new' is the operator, and 'data-type' defines the type of the data.**

**Example 1:**

1. **int \*p;**
2. **p = new int;**

**In the above example, 'p' is a pointer of type int.**

**Example 2:**

1. **float \*q;**
2. **q = new float;**

**In the above example, 'q' is a pointer of type float.**

**In the above case, the declaration of pointers and their assignments are done separately. We can also combine these two statements as follows:**

1. **int \*p = new int;**
2. **float \*q =   new float;**

### Assigning a value to the newly created object

**Two ways of assigning values to the newly created object:**

* **We can assign the value to the newly created object by simply using the assignment operator. In the above case, we have created two pointers 'p' and 'q' of type int and float, respectively. Now, we assign the values as follows:**

1. **\*p = 45;**
2. **\*q = 9.8;**

**We assign 45 to the newly created int object and 9.8 to the newly created float object.**

* **We can also assign the values by using new operator which can be done as follows:**

1. **pointer\_variable = new data-type(value);**

**Let's look at some examples.**

1. **int \*p = new int(45);**
2. **float \*p = new float(9.8);**

### How to create a single dimensional array

**As we know that new operator is used to create memory space for any data-type or even user-defined data type such as an array, structures, unions, etc., so the syntax for creating a one-dimensional array is given below:**

1. **pointer-variable = new data-type[size];**

### Examples:

1. **int \*a1 = new int[8];**

**In the above statement, we have created an array of type int having a size equal to 8 where p[0] refers first element, p[1] refers the first element, and so on.**

### Delete operator

**When memory is no longer required, then it needs to be deallocated so that the memory can be used for another purpose. This can be achieved by using the delete operator, as shown below:**

1. **delete pointer\_variable;**

**In the above statement, 'delete' is the operator used to delete the existing object, and 'pointer\_variable' is the name of the pointer variable.**

**In the previous case, we have created two pointers 'p' and 'q' by using the new operator, and can be deleted by using the following statements:**

1. **delete p;**
2. **delete q;**

**The dynamically allocated array can also be removed from the memory space by using the following syntax:**

1. **delete [size] pointer\_variable;**

**In the above statement, we need to specify the size that defines the number of elements that are required to be freed. The drawback of this syntax is that we need to remember the size of the array. But, in recent versions of C++, we do not need to mention the size as follows:**

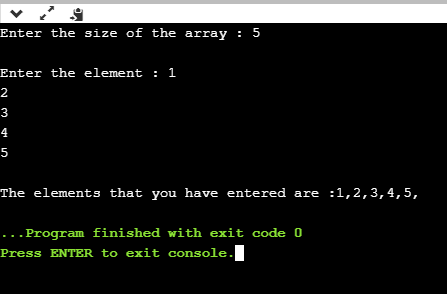
1. **delete [ ] pointer\_variable;**

**Let's understand through a simple example:**

1. **#include <iostream>**
2. **using namespace std**
3. **int main()**
4. **{**
5. **int size;  // variable declaration**
6. **int \*arr = new int[size];   // creating an array**
7. **cout<<"Enter the size of the array : ";**
8. **std::cin >> size;    //**
9. **cout<<"\nEnter the element : ";**
10. **for(int i=0;i<size;i++)   // for loop**
11. **{**
12. **cin>>arr[i];**
13. **}**
14. **cout<<"\nThe elements that you have entered are :";**
15. **for(int i=0;i<size;i++)    // for loop**
16. **{**
17. **cout<<arr[i]<<",";**
18. **}**
19. **delete arr;  // deleting an existing array.**
20. **return 0;**
21. **}**

**In the above code, we have created an array using the new operator. The above program will take the user input for the size of an array at the run time. When the program completes all the operations, then it deletes the object by using the statement delete arr.**

**Output**

****

### Advantages of the new operator

**The following are the advantages of the new operator over malloc() function:**

* **It does not use the sizeof() operator as it automatically computes the size of the data object.**
* **It automatically returns the correct data type pointer, so it does not need to use the typecasting.**
* **Like other operators, the new and delete operator can also be overloaded.**
* **It also allows you to initialize the data object while creating the memory space for the object.**

**self containing class**

**Syntax for Containership:**

**// Class that is to be contained**

**class first {**

**.**

**.**

**};**

**// Container class**

**class second {**

**// creating object of first**

**first f;**

**.**

**.**

**};**

**Below examples explain the Containership in C++ in a better way.**

**Example 1:**

|  |
| --- |
| **// CPP program to illustrate**  **// concept of Containership**    **#include <iostream>**  **using namespace std;**    **class first {**  **public:**  **void showf()**  **{**  **cout << "Hello from first class\n";**  **}**  **};**    **// Container class**  **class second {**  **// creating object of first**  **first f;**    **public:**  **// constructor**  **second()**  **{**  **// calling function of first class**  **f.showf();**  **}**  **};**    **int main()**  **{**  **// creating object of second**  **second s;**  **}** |

**Output:**

**Hello from first class**

**Explanation:In the class second we have an object of class first. This is another type of inheritance we are witnessing. This type of inheritance is known as has\_a relationship as we say that class second has an object of first class first as its member. From the object f we call the function of class first.**

**Example 2:**

|  |
| --- |
| **#include <iostream>**  **using namespace std;**    **class first {**  **public:**  **first()**  **{**  **cout << "Hello from first class\n";**  **}**  **};**    **// Container class**  **class second {**  **// creating object of first**  **first f;**    **public:**  **// constructor**  **second()**  **{**  **cout << "Hello from second class\n";**  **}**  **};**    **int main()**  **{**  **// creating object of second**  **second s;**  **}** |

**Output:**

**Hello from first class**

**Hello from second class**

**Explanation:In this program we have not inherited class first into class second but as we are having an object of class first as a member of class second. So when default constructor of class second is called, due to presence of object f of first class in second, default constructor of class first is called first and then default constructor of class second is called .**

**C++ this Pointer**

**In C++ programming, this is a keyword that refers to the current instance of the class. There can be 3 main usage of this keyword in C++.**

* **It can be used to pass current object as a parameter to another method.**
* **It can be used to refer current class instance variable.**
* **It can be used to declare indexers.**

**C++ this Pointer Example**

**Let's see the example of this keyword in C++ that refers to the fields of current class.**

1. **#include <iostream>**
2. **using namespace std;**
3. **class Employee {**
4. **public:**
5. **int id; //data member (also instance variable)**
6. **string name; //data member(also instance variable)**
7. **float salary;**
8. **Employee(int id, string name, float salary)**
9. **{**
10. **this->id = id;**
11. **this->name = name;**
12. **this->salary = salary;**
13. **}**
14. **void display()**
15. **{**
16. **cout<<id<<"  "<<name<<"  "<<salary<<endl;**
17. **}**
18. **};**
19. **int main(void) {**
20. **Employee e1 =Employee(101, "Sonoo", 890000); //creating an object of Employee**
21. **Employee e2=Employee(102, "Nakul", 59000); //creating an object of Employee**
22. **e1.display();**
23. **e2.display();**
24. **return 0;**
25. **}**

**Output:**

**101 Sonoo 890000**

**102 Nakul 59000**

**C++ virtual function**

* **A C++ virtual function is a member function in the base class that you redefine in a derived class. It is declared using the virtual keyword.**
* **It is used to tell the compiler to perform dynamic linkage or late binding on the function.**
* **There is a necessity to use the single pointer to refer to all the objects of the different classes. So, we create the pointer to the base class that refers to all the derived objects. But, when base class pointer contains the address of the derived class object, always executes the base class function. This issue can only be resolved by using the 'virtual' function.**
* **A 'virtual' is a keyword preceding the normal declaration of a function.**
* **When the function is made virtual, C++ determines which function is to be invoked at the runtime based on the type of the object pointed by the base class pointer.**

**Late binding or Dynamic linkage**

**In late binding function call is resolved during runtime. Therefore compiler determines the type of object at runtime, and then binds the function call.**

**Rules of Virtual Function**

* **Virtual functions must be members of some class.**
* **Virtual functions cannot be static members.**
* **They are accessed through object pointers.**
* **They can be a friend of another class.**
* **A virtual function must be defined in the base class, even though it is not used.**
* **The prototypes of a virtual function of the base class and all the derived classes must be identical. If the two functions with the same name but different prototypes, C++ will consider them as the overloaded functions.**
* **We cannot have a virtual constructor, but we can have a virtual destructor**
* **Consider the situation when we don't use the virtual keyword.**

1. **#include <iostream>**
2. **using namespace std;**
3. **class A**
4. **{**
5. **int x=5;**
6. **public:**
7. **void display()**
8. **{**
9. **std::cout << "Value of x is : " << x<<std::endl;**
10. **}**
11. **};**
12. **class B: public A**
13. **{**
14. **int y = 10;**
15. **public:**
16. **void display()**
17. **{**
18. **std::cout << "Value of y is : " <<y<< std::endl;**
19. **}**
20. **};**
21. **int main()**
22. **{**
23. **A \*a;**
24. **B b;**
25. **a = &b;**
26. **a->display();**
27. **return 0;**
28. **}**

**Output:**

**Value of x is : 5**

**In the above example, \* a is the base class pointer. The pointer can only access the base class members but not the members of the derived class. Although C++ permits the base pointer to point to any object derived from the base class, it cannot directly access the members of the derived class. Therefore, there is a need for virtual function which allows the base pointer to access the members of the derived class.**

**How to find Nth Highest Salary in SQL**

**C++ virtual function Example**

**Let's see the simple example of C++ virtual function used to invoked the derived class in a program.**

1. **#include <iostream>**
2. **{**
3. **public:**
4. **virtual void display()**
5. **{**
6. **cout << "Base class is invoked"<<endl;**
7. **}**
8. **};**
9. **class B:public A**
10. **{**
11. **public:**
12. **void display()**
13. **{**
14. **cout << "Derived Class is invoked"<<endl;**
15. **}**
16. **};**
17. **int main()**
18. **{**
19. **A\* a;    //pointer of base class**
20. **B b;     //object of derived class**
21. **a = &b;**
22. **a->display();   //Late Binding occurs**
23. **}**

**Output:**

**Derived Class is invoked**

**Pure Virtual Function**

* **A virtual function is not used for performing any task. It only serves as a placeholder.**
* **When the function has no definition, such function is known as "do-nothing" function.**
* **The "do-nothing" function is known as a pure virtual function. A pure virtual function is a function declared in the base class that has no definition relative to the base class.**
* **A class containing the pure virtual function cannot be used to declare the objects of its own, such classes are known as abstract base classes.**
* **The main objective of the base class is to provide the traits to the derived classes and to create the base pointer used for achieving the runtime polymorphism.**

**Pure virtual function can be defined as:**

1. **virtual void display() = 0;**

**Let's see a simple example:**

1. **#include <iostream>**
2. **using namespace std;**
3. **class Base**
4. **{**
5. **public:**
6. **virtual void show() = 0;**
7. **};**
8. **class Derived : public Base**
9. **{**
10. **public:**
11. **void show()**
12. **{**
13. **std::cout << "Derived class is derived from the base class." << std::endl;**
14. **}**
15. **};**
16. **int main()**
17. **{**
18. **Base \*bptr;**
19. **//Base b;**
20. **Derived d;**
21. **bptr = &d;**
22. **bptr->show();**
23. **return 0;**
24. **}**

**Output:**

**Derived class is derived from the base class.**

**In the above example, the base class contains the pure virtual function. Therefore, the base class is an abstract base class. We cannot create the object of the base class.**

**C++ Abstract class**

**In C++ class is made abstract by declaring at least one of its functions as <>strong>pure virtual function. A pure virtual function is specified by placing "= 0" in its declaration. Its implementation must be provided by derived classes.**

**Let's see an example of abstract class in C++ which has one abstract method draw(). Its implementation is provided by derived classes: Rectangle and Circle. Both classes have different implementation.**

**Exception Handling in Java - Javatpoint**

1. **#include <iostream>**
2. **using namespace std;**
3. **class Shape**
4. **{**
5. **public:**
6. **virtual void draw()=0;**
7. **};**
8. **class Rectangle : Shape**
9. **{**
10. **public:**
11. **void draw()**
12. **{**
13. **cout < <"drawing rectangle..." < <endl;**
14. **}**
15. **};**
16. **class Circle : Shape**
17. **{**
18. **public:**
19. **void draw()**
20. **{**
21. **cout <<"drawing circle..." < <endl;**
22. **}**
23. **};**
24. **int main( ) {**
25. **Rectangle rec;**
26. **Circle cir;**
27. **rec.draw();**
28. **cir.draw();**
29. **return 0;**
30. **}**

**Output:**

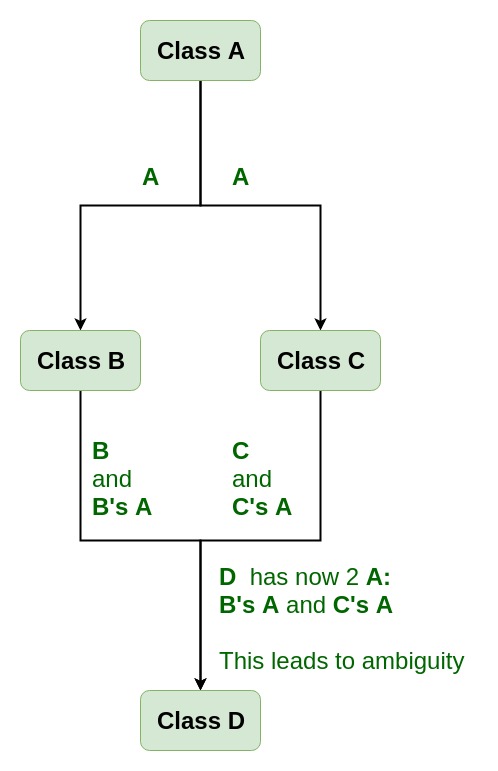
**drawing rectangle...**

**drawing circle...**

**Virtual base class in C++**

**Virtual base classes are used in virtual inheritance in a way of preventing multiple “instances” of a given class appearing in an inheritance hierarchy when using multiple inheritances.**

**Need for Virtual Base Classes:  
Consider the situation where we have one class A .This class is A is inherited by two other classes B and C. Both these class are inherited into another in a new class D as shown in figure below.**

****

**As we can see from the figure that data members/function of class A are inherited twice to class D. One through class B and second through class C. When any data / function member of class A is accessed by an object of class D, ambiguity arises as to which data/function member would be called? One inherited through B or the other inherited through C. This confuses compiler and it displays error.**

**Example: To show the need of Virtual Base Class in C++**

|  |
| --- |
| **#include <iostream>**  **using namespace std;**    **class A {**  **public:**  **void show()**  **{**  **cout << "Hello form A \n";**  **}**  **};**    **class B : public A {**  **};**    **class C : public A {**  **};**    **class D : public B, public C {**  **};**    **int main()**  **{**  **D object;**  **object.show();**  **}** |

**Compile Errors:**

**prog.cpp: In function 'int main()':**

**prog.cpp:29:9: error: request for member 'show' is ambiguous**

**object.show();**

**^**

**prog.cpp:8:8: note: candidates are: void A::show()**

**void show()**

**^**

**prog.cpp:8:8: note: void A::show()**

**How to resolve this issue?**

**To resolve this ambiguity when class A is inherited in both class B and class C, it is declared as virtual base class by placing a keyword virtual as :**

**Syntax for Virtual Base Classes:**

**Syntax 1:**

**class B : virtual public A**

**{**

**};**

**Syntax 2:**

**class C : public virtual A**

**{**

**};**

**Note: virtual can be written before or after the public. Now only one copy of data/function member will be copied to class C and class B and class A becomes the virtual base class.  
Virtual base classes offer a way to save space and avoid ambiguities in class hierarchies that use multiple inheritances. When a base class is specified as a virtual base, it can act as an indirect base more than once without duplication of its data members. A single copy of its data members is shared by all the base classes that use virtual base.**

**Example 1**

|  |
| --- |
| **#include <iostream>**  **using namespace std;**    **class A {**  **public:**  **int a;**  **A() // constructor**  **{**  **a = 10;**  **}**  **};**    **class B : public virtual A {**  **};**    **class C : public virtual A {**  **};**    **class D : public B, public C {**  **};**    **int main()**  **{**  **D object; // object creation of class d**  **cout << "a = " << object.a << endl;**    **return 0;**  **}** |

**Output:**

**a = 10**

**Explanation :The class A has just one data member a which is public. This class is virtually inherited in class B and class C. Now class B and class C becomes virtual base class and no duplication of data member a is done.**

**Example 2:**

|  |
| --- |
| **#include <iostream>**  **using namespace std;**    **class A {**  **public:**  **void show()**  **{**  **cout << "Hello from A \n";**  **}**  **};**    **class B : public virtual A {**  **};**    **class C : public virtual A {**  **};**    **class D : public B, public C {**  **};**    **int main()**  **{**  **D object;**  **object.show();**  **}** |

**Output:**

**Hello from A**

**What is a friend function?**

**A friend function is a function of the class defined outside the class scope but it has the right to access all the private and protected members of the class.**

**The friend functions appear in the class definition but friends are the member functions.**

**Characteristics of a Friend function:**

* **The function is not in the scope of the class to which it has been declared as a friend.**
* **It cannot be called using the object as it is not in the scope of that class.**
* **It can be invoked like a normal function without using the object.**
* **It cannot access the member names directly and has to use an object name and dot membership operator with the member name.**
* **It can be declared either in the private or the public part.**

**Why do we need a friend function in C++?**

* **Friend function in C++ is used when the class private data needs to be accessed directly without using object of that class.**
* **Friend functions are also used to perform operator overloading. As we already know about the function overloading, operators can also be overloaded with the help of operator overloading.**

**Characteristics of a Friend function**

**HTML Tutorial**

* **The friend function is declared using friend keyword.**
* **It is not a member of the class but it is a friend of the class.**
* **As it is not a member of the class so it can be defined like a normal function.**
* **Friend functions do not access the class data members directly but they pass an object as an argument.**
* **It is like a normal function.**
* **If we want to share the multiple class's data in a function then we can use the friend function.**

**Syntax for the declaration of a friend function.**

1. **class class\_name**
2. **{**
3. **friend data\_type function\_name(argument/s);            // syntax of friend function**
4. **};**

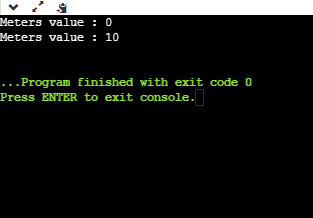
**In the above declaration, the friend function is preceded by the keyword friend. The function can be defined anywhere in the program like a normal C++ function. The function definition does not use either the keyword friend or scope resolution operator.**

**Let's understand the friend function through an example.**

1. **#include <iostream>**
2. **using namespace std;**
3. **class Distance**
4. **{**
5. **private:**
6. **int meters;**
7. **public:**
8. **// constructor**
9. **Distance()**
10. **{**
11. **meters = 0;**
12. **}**
13. **// definition of display\_data() method**
14. **void display\_data()**
15. **{**
16. **std::cout << "Meters value : " << meters<<std::endl;**
17. **}**
19. **//prototype of a friend function.**
20. **friend void addvalue(Distance &d);**
22. **};**
23. **// Definition of friend function**
24. **void addvalue(Distance &d) //  argument contain the reference**
25. **{**
26. **d.meters = d.meters+10; // incrementing the value of meters by 10.**
27. **}**
28. **// main() method**
29. **int main()**
30. **{**
31. **Distance d1; // creating the object of class distance.**
32. **d1.display\_data(); // meters = 0**
33. **addvalue(d1); // calling friend function**
34. **d1.display\_data(); // meters = 10**
35. **return 0;**
36. **}**

**In the above code, Distance is the class that contains private field named as 'meters'. The Distance() is the constructor method that initializes the 'meters' value with 0. The display\_data() is a method that displays the 'meters' value. The addvalue() is a friend function of Distance class that modifies the value of 'meters'. Inside the main() method, d1 is an object of a Distance class.**

**Output**

****

**Friend function can also be useful when we are working on objects of two different classes.**

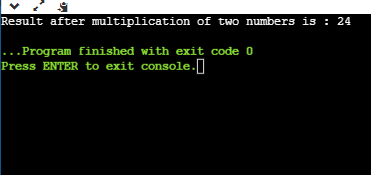
**Let's understand through an example.**

1. **// Add members of two different classes using friend functions**
2. **#include <iostream>**
3. **using namespace std;**
4. **// forward declaration of a class**
5. **class ClassB;**
7. **// declaration of a class**
8. **class ClassA {**
9. **public:**
10. **// constructor ClassA() to initialize num1 to 12**
11. **ClassA()**
12. **{**
13. **num1 =12;**
15. **}**
17. **private:**
18. **int num1;  // declaration of integer variable**
20. **// friend function declaration**
21. **friend int multiply(ClassA, ClassB);**
22. **};**
23. **class ClassB {**
24. **public:**
25. **// constructor ClassB() to initialize num2 to 2**
26. **ClassB()**
27. **{**
28. **num2 = 2;**
29. **}**
30. **private:**
31. **int num2;  // declaration of integer variable**
32. **// friend function declaration**
33. **friend int multiply(ClassA, ClassB);**
34. **};**
36. **// access members of both classes**
37. **int multiply(ClassA object1, ClassB object2)**
38. **{**
39. **return (object1.num1 \* object2.num2);**
40. **}**
42. **int main() {**
43. **ClassA object1;   // declaration of object of ClassA**
44. **ClassB object2;   // declaration of object of ClassB**
45. **cout << "Result after multiplication of two numbers is : " << multiply(object1, object2);**
46. **return 0;**
47. **}**

**In the above code, we have defined two classes named as ClassA and ClassB. Both these classes contain the friend function 'multiply()'. The friend function can access the data members of both the classes, i.e., ClassA and ClassB. The multiply() function accesses the num1 and num2 of ClassA and ClassB respectively. In the above code, we have created object1 and object2 of ClassA and ClassB respectively. The multiply() function multiplies the num1 and num2 and returns the result.**

**As we can observe in the above code that the friend function in ClassA is also using ClassB without prior declaration of ClassB. So, in this case, we need to provide the forward declaration of ClassB.**

**Output**

****

**Friend class in C++**

**We can also create a friend class with the help of friend keyword.**

1. **class Class1;**
2. **class Class2**
3. **{**
4. **// Class1 is a friend class of Class2**
5. **friend class Class1;**
6. **.. .....**
7. **}**
8. **class Class1**
9. **{**
10. **....**
11. **}**

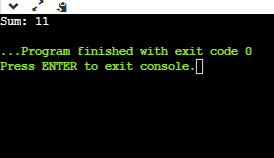
**In the above declaration, Class1 is declared as a friend class of Class2. All the members of Class2 can be accessed in Class1.**

**Let's understand through an example.**

1. **// C++ program to demonstrate the working of friend class**
2. **#include <iostream>**
3. **using namespace std;**
4. **// forward declaration**
5. **class ClassB;**
7. **class ClassA {**
8. **private:**
9. **int num1;**
11. **// friend class declaration**
12. **friend class ClassB;**
14. **public:**
15. **// constructor to initialize numA to 10**
16. **ClassA()**
17. **{**
18. **num1 = 10;**
19. **}**
20. **};**
22. **class ClassB {**
23. **private:**
24. **int num2;**
26. **public:**
27. **// constructor to initialize numB to 1**
28. **ClassB()**
29. **{**
30. **num2 = 1;**
31. **}**
33. **// member function to add num1**
34. **// from ClassA and num2 from ClassB**
35. **int add() {**
36. **ClassA objectA;**
37. **return objectA.num1 + num2;**
38. **}**
39. **};**
41. **int main() {**
42. **ClassB objectB;**
43. **cout << "Sum: " << objectB.add();**
44. **return 0;**
45. **}**

**In the above code, we have created two classes named as ClassA and ClassB. Since ClassA is declared as friend of ClassB, so ClassA can access all the data members of ClassB. In ClassB, we have defined a function add() that returns the sum of num1 and num2. Since ClassB is declared as friend of ClassA, so we can create the objects of ClassA in ClassB.**

**Output**

****

# Static Member Function in C++

**The static is a keyword in the C and C++ programming language. We use the static keyword to define the static data member or static member function inside and outside of the class. Let's understand the static data member and static member function using the programs.**

## Static data member

**When we define the data member of a class using the static keyword, the data members are called the static data member. A static data member is similar to the static member function because the static data can only be accessed using the static data member or static member function. And, all the objects of the class share the same copy of the static member to access the static data.**

**Syntax**

1. **static data\_type data\_member;**

**Here, the static is a keyword of the predefined library.**

**Features of Java - Javatpoint**

**The data\_type is the variable type in C++, such as int, float, string, etc.**

**The data\_member is the name of the static data.**

**Example 1: Let's create a simple program to access the static data members in the C++ programming language.**

1. **#include <iostream>**
2. **#include <string.h>**
3. **using namespace std;**
4. **// create class of the Car**
5. **class Car**
6. **{**
7. **private:**
8. **int car\_id;**
9. **char car\_name[20];**
10. **int marks;**
12. **public:**
13. **// declare a static data member**
14. **static int static\_member;**
16. **Car()**
17. **{**
18. **static\_member++;**
19. **}**
21. **void inp()**
22. **{**
23. **cout << " \n\n Enter the Id of the Car: " << endl;**
24. **cin >> car\_id; // input the id**
25. **cout << " Enter the name of the Car: " << endl;**
26. **cin  >> car\_name;**
27. **cout << " Number of the Marks (1 - 10): " << endl;**
28. **cin >> marks;**
29. **}**
31. **// display the entered details**
32. **void disp ()**
33. **{**
34. **cout << " \n Id of the Car: " << car\_id;**
35. **cout << "\n Name of the Car: " << car\_name;**
36. **cout << " \n Marks: " << marks;**
38. **}**
39. **};**
41. **// initialized the static data member to 0**
42. **int Car::static\_member = 0;**
44. **int main ()**
45. **{**
46. **// create object for the class Car**
47. **Car c1;**
48. **// call inp() function to insert values**
49. **c1. inp ();**
50. **c1. disp();**
52. **//create another object**
53. **Car c2;**
54. **// call inp() function to insert values**
55. **c2. inp ();**
56. **c2. disp();**

59. **cout << " \n No. of objects created in the class: " << Car :: static\_member <<endl;**
60. **return 0;**
61. **}**

**Output**

**Enter the Id of the Car:**

**101**

**Enter the name of the Car:**

**Ferrari**

**Number of the Marks (1 - 10):**

**10**

**Id of the Car: 101**

**Name of the Car: Ferrari**

**Marks: 10**

**Enter the Id of the Car:**

**205**

**Enter the name of the Car:**

**Mercedes**

**Number of the Marks (1 - 10):**

**9**

**Id of the Car: 205**

**Name of the Car: Mercedes**

**Marks: 9**

**No. of objects created in the class: 2**

### Static Member Functions

**The static member functions are special functions used to access the static data members or other static member functions. A member function is defined using the static keyword. A static member function shares the single copy of the member function to any number of the class' objects. We can access the static member function using the class name or class' objects. If the static member function accesses any non-static data member or non-static member function, it throws an error.**

**Syntax**

1. **class\_name::function\_name (parameter);**

**Here, the class\_name is the name of the class.**

**function\_name: The function name is the name of the static member function.**

**parameter: It defines the name of the pass arguments to the static member function.**

**Example 2: Let's create another program to access the static member function using the class name in the C++ programming language.**

1. **#include <iostream>**
2. **using namespace std;**
3. **class Note**
4. **{**
5. **// declare a static data member**
6. **static int num;**
8. **public:**
9. **// create static member function**
10. **static int func ()**
11. **{**
12. **return num;**
13. **}**
14. **};**
15. **// initialize the static data member using the class name and the scope resolution operator**
16. **int Note :: num = 5;**
18. **int main ()**
19. **{**
20. **// access static member function using the class name and the scope resolution**
21. **cout << " The value of the num is: " << Note:: func () << endl;**
22. **return 0;**
23. **}**

**Output**

**The value of the num is: 5**

**Example 3: Let's create another program to access the static member function using the class' object in the C++ programming language.**

1. **#include <iostream>**
2. **using namespace std;**
3. **class Note**
4. **{**
5. **// declare a static data member**
6. **static int num;**
8. **public:**
9. **// create static member function**
10. **static int func ()**
11. **{**
12. **cout << " The value of the num is: " << num << endl;**
13. **}**
14. **};**
15. **// initialize the static data member using the class name and the scope resolution operator**
16. **int Note :: num = 15;**
18. **int main ()**
19. **{**
20. **// create an object of the class Note**
21. **Note n;**
22. **// access static member function using the object**
23. **n.func();**
25. **return 0;**
26. **}**

**Output**

**The value of the num is: 15**

**Example 4: Let's consider an example to access the static member function using the object and class in the C++ programming language.**

1. **#include <iostream>**
2. **using namespace std;**
3. **class Member**
4. **{**
6. **private:**
7. **// declaration of the static data members**
8. **static int A;**
9. **static int B;**
10. **static int C;**
12. **// declare public access specifier**
13. **public:**
14. **// define the static member function**
15. **static void disp ()**
16. **{**
17. **cout << " The value of the A is: " << A << endl;**
18. **cout << " The value of the B is: " << B << endl;**
19. **cout << " The value of the C is: " << C << endl;**
20. **}**
21. **};**
22. **// initialization of the static data members**
23. **int Member :: A = 20;**
24. **int Member :: B = 30;**
25. **int Member :: C = 40;**
27. **int main ()**
28. **{**
29. **// create object of the class Member**
30. **Member mb;**
31. **// access the static member function using the class object name**
32. **cout << " Print the static member through object name: " << endl;**
33. **mb. disp();**
34. **// access the static member function using the class name**
35. **cout << " Print the static member through the class name: " << endl;**
36. **Member::disp();**
37. **return 0;**
38. **}**

**Output**

**Print the static member through object name:**

**The value of the A is: 20**

**The value of the B is: 30**

**The value of the C is: 40**

**Print the static member through the class name:**

**The value of the A is: 20**

**The value of the B is: 30**

**The value of the C is: 40**